CS61B, Fall 2014 **Project #1 (revision 2)** **P. N. Hilfinger**

Due: Wednesday, 22 October 2014 at 2400

1 Introduction

This project involves writing a miniature *relational database management system (DBMS)* that stores *tables* of data, where a table consists of some number of labeled *columns* of information. Our system will include a very simple *query language* for extracting information from these tables. For the purposes of this project, we will deal only with very small databases, and therefore will not consider speed and efficiency at all. For that sort of stuff, you might consider taking CS186 at some point.

As an example, consider the following set of three tables containing information about students, about course numbers and locations, and about grades in these courses. Each table has a name (given above the upper-left corner) and each column of each table has a name (given above the double line).

students

SID	**Lastname**	**Firstname**	**SemEnter**	**YearEnter**	**Major**
101	Knowles	Jason	F	2003	EECS
102	Chan	Valerie	S	2003	Math
103	Xavier	Jonathan	S	2004	LSUnd
104	Armstrong	Thomas	F	2003	EECS
105	Brown	Shana	S	2004	EECS
106	Chan	Yangfan	F	2003	LSUnd

schedule

CCN	**Num**	**Dept**	**Time**	**Room**	**Sem**	**Year**
21228	61A	EECS	2-3MWF	1 Pimentel	F	2003
21231	61A	EECS	1-2MWF	1 Pimentel	S	2004
21229	61B	EECS	11-12MWF	155 Dwinelle	F	2003
21232	61B	EECS	1-2MWF	2050 VLSB	S	2004
21103	54	Math	1-2MWF	2050 VLSB	F	2003
21105	54	Math	1-2MWF	1 Pimentel	S	2004
21001	1A	English	9-10MWF	2301 Tolman	F	2003
21005	1A	English	230-5TuTh	130 Wheeler	S	2004

enrolled

SID	**CCN**	**Grade**
101	21228	B
101	21105	B+
101	21232	A-
101	21001	B
102	21231	A
102	21105	A-
102	21229	A
102	21001	B+
103	21105	B+
103	21005	B+
104	21228	A-
104	21229	B+
104	21105	A-
104	21005	A-
105	21228	A
105	21001	B+
106	21103	A
106	21001	B
106	21231	A

2 Describing Command Syntax

You communicate with the database system using an artificial notation usually known as a *language*, although it is much simpler than any human language. The definition and processing of such languages is an important skill for any computer scientists. We normally think of programming languages such as Java, but there are many other contexts where small, *domain-specific languages* (DSLs) are appropriate engineering solutions to a design or implementation problem.

We typically describe the *syntax* (grammatical structure) of a language using a specialized *metalanguage*—a language for describing languages. Here, we'll use a version of a common metalanguage: BNF (Backus-Naur Form). A *BNF grammar* consists of a set of rules, For example:

$\langle$*create statement*$\rangle$ ::= `create table` $\langle$*name*$\rangle$ $\langle$*table definition*$\rangle$
$\langle$*table definition*$\rangle$ ::=
  `(` $\langle\textit{name}\rangle^{+}_{,}$ `)`
 | `as` $\langle$*select clause*$\rangle$

means that a *create statement* consists of the (literal) words "create" and "table" followed by a *name*, followed by a *table definition.* A *table definition*, in turn, consists of either

- a left parenthesis, followed by a list of one or more *names* separated by commas, followed by a right parenthesis; or
- the word "as" followed by a *select clause* (which is defined in another rule.)

The labels bracketed by `<>` are *metavariables* and stand for sets of possible strings, as described by the rules.

One other notation is useful:

$\langle$*program*$\rangle$ ::= $\langle\textit{statement}\rangle^{*}$

means that a *program* consists of zero or more *statements.*

We define a few of the metavariables in English (sort of as base cases):

`<`*name*`>` denotes a sequence of letters, digits, and underscores that does not start with a digit.

`<`*literal*`>` denotes a sequence of zero or more characters other than ends-of-lines, commas, or single quotes (apostrophes), surrounded by single quotes. For example,

```
'Mary Smith'
```

`<`*empty*`>` The empty string: stands for a missing clause.

As is traditional, we ignore whitespace (spaces, tabs, newlines) in the descriptions that follow. Whitespace or other punctuation must separate words from each other. For example, "`createtable`" is just a name, and not the two words `create` and `table`. Comments, which are the same as `/* */` comments in Java, are treated as whitespace.

3 Commands

Our database system uses a very restricted dialect of SQL (Structured Query Language), a widely used notation for communicating with relational databases. When you run the database system, it will accept a sequence of commands from the standard input (i.e., normally the terminal), according to the following syntax:

<*program*> ::= <*statement*>*
<*statement*> ::=
  <*create statement*>
 | <*exit statement*>
 | <*insert statement*>
 | <*load statement*>
 | <*print statement*>
 | <*select statement*>
 | <*store statement*>
<*create statement*> ::= `create table` <*name*> <*table definition*> `;`
<*table definition*> ::=
  `(` <*column name*>$^+_{,}$ `)`
 | `as` <*select clause*>
<*print statement*> ::= `print` <*table name*> `;`
<*insert statement*> ::= `insert into` <*table name*> `values` <*literal*>$^+_{,}$ `;`
<*load statement*> ::= `load` <*name*> `;`
<*store statement*> ::= `store` <*table name*> `;`
<*exit statement*> ::= `quit ;` | `exit ;`
<*select statement*> ::= <*select clause*> `;`
<*select clause*> ::= `select` <*column name*>$^+_{,}$ `from` <*tables*> <*condition clause*>
<*condition clause*> ::=
  `where` <*condition*>$^+_{\texttt{and}}$
 | <*empty*>
<*tables*> ::= <*table name*> | <*table name*> `,` <*table name*>
<*condition*> ::=
  <*column name*> <*relation*> <*column name*>
 | <*column name*> <*relation*> <*literal*>
<*relation*> ::= `<` | `>` | `=` | `!=` | `<=` | `>=`
<*table name*> ::= <*name*>
<*column name*> ::= <*name*>

The above only defines the syntax, but doesn't say what these statements do (known as the *semantics*). For that, we just use English:

create table *table name* `(` <*column name*>$^+_{,}$ `)` Creates an empty table with the given name (replacing it if it is already loaded). The names of its columns are given by the *column names* in order. There must not be any duplicate column names.

create table *table* **as** <*select clause*>

Creates a table with the given name (replacing it if it is already loaded) whose columns and contents are those produced by the *select clause*. Each distinct set of column values gets only one row in the table (no duplicate rows).

load <*name*>**;**
Load data from the file *name*`.db` to create a table named *table*.

store <*table name*>**;**
Store the data from the table *table name* into the file *table name*`.db`.

insert into <*table name*> `values` $<\textit{literal}>^{+}_{,}$ **;**
Add a new row to the given *table* whose values are given by the list of literals. There must be exactly one literal for each column of the table, and the table must already exist. This command has no effect if there is already a row in the table with these values.

print <*table name*> **;**
Print all rows of the table with the given name (which must be loaded). The rows are printed one per line and indented. Separate columns with blanks, and print the columns in the order they were specified when the table was created. See the example below for the format.

<*select clause*>`;` Select clauses are described below. They represent tables created from other tables. When used alone as a statement (terminated by a semicolon), they indicate that the resulting table is to be printed, using the format described above for the `print` command.

quit ;
Exit the program.

exit ;
Synonym for `quit`.

Select clauses. Select clauses are used in `select` statements and in `create` statements. They denote tables whose information is selected from other tables.

`select` $<\textit{column name}>^{+}_{,}$ `from` <*table name*> <*condition clause*> A new (unnamed) table consisting of the named columns from the given table from all rows that satisfy the *condition clause.*

`select` $<\textit{column name}>^{+}_{,}$ `from` <*table name*> `,` <*table name*> <*condition clause*> A new (unnamed) table consisting of the given columns from all rows of the *natural inner join* of the two tables that satisfy the *condition clause.* A natural inner join is a table whose rows are formed by combining pairs of rows, one from the first table and one from the second, such that any columns that have the same name in both tables also have the same value.

Condition clauses. An empty *condition clause* does not restrict the rows that appear in a `select`. Otherwise, it contains one or more *conditions* separated by `and`, all of which must be satisfied. The tests compare column values against either other columns or literal values. The relation symbols mean what they do in Java, except that all values are treated as strings (use the `compareTo` method on Strings to compare them). Thus you can write things like

```
Lastname >= 'Chan'
```

to get all rows in which the value of the `Lastname` column comes after "Chan" in "lexicographic order." This is roughly like dictionary order, with the major difference being that all digits come before all capital letters, which come before all lower-case letters.

4 Format of .db Files

A `.db` file starts with a line containing all column names (at least one) separated by commas, with any leading or trailing whitespace removed (see the `.split` and `.trim` methods of the `String` class). Column names must be valid identifiers and must be distinct. This is followed by any number of lines (zero of more), one for each row, containing the values for that row, separated by commas (again, leading or trailing whitespace is removed). For example, the 'students' table shown previously would look like this in a file:

```
SID,Lastname,Firstname,SemEnter,YearEnter,Major
101,Knowles,Jason,F,2003,EECS
102,Chan,Valerie,S,2003,Math
103,Xavier,Jonathan,S,2004,LSUnd
104,Armstrong,Thomas,F,2003,EECS
105,Brown,Shana,S,2004,EECS
106,Chan,Yangfan,F,2003,LSUnd
```

5 Example

If the information in these tables exists in three files—`students.db`, `schedule.db`, and `enrolled.db`—then a session with our DBMS might look like the following transcript. Characters typed by the user are underlined.

```
DB61B System.  Version 1.0
> load students ;
Loaded students.db
> load enrolled ;
Loaded enrolled.db
> load schedule ;
Loaded schedule.db
> /* What are the names and SIDS of all students whose last name
     is 'Chan'?  */
```

```
> select SID, Firstname from students
     where Lastname = 'Chan';
Search results:
  102 Valerie
  106 Yangfan
> /* Who took the course with CCN 21001, and what were their grades?  */
> select Firstname, Lastname, Grade
        from students, enrolled where CCN = '21001';
Search results:
  Jason Knowles B
  Shana Brown B+
  Yangfan Chan B
  Valerie Chan B+
> /* Who has taken the course named 61A from EECS? */
> /* First, create a table that contains SIDs and course names */
> create table enrolled2 as select SID
     from enrolled, schedule
     where Dept = 'EECS' and Num = '61A';
> /* Print these SIDs */
> print enrolled2;
Contents of enrolled2:
  101
  102
  104
  105
  106
> /* Now print the names of the students in this list */
> select Firstname, Lastname from students, enrolled2;
Search results:
  Jason Knowles
  Valerie Chan
  Thomas Armstrong
  Shana Brown
  Yangfan Chan
> quit ;
```

6 Your Task

The directory `~cs61b/hw/proj1` will contain skeleton files that suggest a structure for this project. Copy them into a fresh directory as a starting point, using the command '`hw init proj1`'. Please read *General Guidelines for Programming Projects* (see the "homework" page on the class web site). To submit your result, use the command '`hw submit proj1`'. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes a syntax error, or mentions a non-existent column, your program should not simply halt and catch fire, but should give some sort of message and then try to get back to a usable state. For syntactic errors (errors in the format of commands) you should skip to the next semicolon (if it hasn't occurred yet) or the end-of-file, whichever comes first. For all errors, you should print a single, separate line starting with the word "error" (in upper or lower case) and followed by any message, and the erroneous command should have no effect on any table. Your program should *not* exit with a cryptic stack trace.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that:

- Your makefile is set up to compile everything on the command `gmake` and to run all your tests on the command `gmake check`. The makefile provided in our skeleton files is already set up to do this. Be sure to keep it up to date if you add additional `.java` files.
- Your main function is in a public class called `db61b.Main`. The skeleton is already set up this way.
- The first line of output of your program identifies the program. It may contain anything.
- Before reading the first command and on reading each subsequent end-of-line or comment, your program must print the prompt '`>␣`' (greater than followed by a blank). or (to indicate a continuation), '`...`'. The testing script will ignore these. The skeleton is set up to do this already.
- Output things in exactly the format shown in the example, with no extra adornment.
- Put your error messages on separate lines, starting with the word '`error`' (upper or lower case). The grading program will ignore the text of the message.
- Your program should exit without further output when it encounters a `quit` (or `exit`) command or an end-of-file in the input.
- Your final version must not print any debugging information.
- When printing out the contents of a table, you need not worry about the order of the rows (the autograder will be clever enough to handle any order). However, you *must* print columns in the order specified in the `.db` file or in the *columns* of the `select` command from which the table came.
- The grading program will ignore extra blank lines and extra blanks at the ends of lines, and will treat any run of consecutive blanks as if it were a single blank.

7 Advice

You will find this project challenging enough without helping to make it harder. Much of what might look complicated to you is actually pretty easy, once you take the time to look to see what has already been written for you—specifically what is in the skeleton and the standard Java library of classes. So, before doing any hard work on anything that looks tricky, browse the skeleton and the documentation.

If you're not using the skeleton, you'll have a couple of extra things worth metnioning to figure out at some point during the project, namely reading input files and choosing a way to store the rows of a table. For reading from files (the `load` command), you'll probably want to use `java.io.FileReader` together with `java.util.Scanner`, which is also good for reading from the standard input. In fact, we've tried to design the syntax to make it particularly easy to parse using `Scanners`. If you find yourself writing lots of complicated junk just to read in and interpret the input, you might back off and ask one of us to look over your approach. The standard Java string type, `java.lang.String`, also contains useful member functions for doing comparisons (including comparisons that ignore case). For choosing a way to store rows in a table, you can use arrays or devise a kind of linked list to do so, but you might instead want to take a look at such interfaces as `java.util.List`, `java.util.Set`, and `java.util.Map` and at the library classes that implement them. Read the on-line documentation for all of these.

It's important to have *something* working as soon as possible. You'll prevent really serious trouble by doing so. I suggest the following order to getting things working:

1. Throughout the project, write test cases (in fact, do this every chance you get). Among other things, writing test cases gets you to understand the specification better. Whenever you find an input that breaks your program, make sure you capture it in a test case.

2. Get the printing of prompts, handling of comments, and the 'quit' and exit' commands, and the end of input to work. If you're using our skeleton, this step has already been completed for you.

3. Implement the Row class, except for the `Row(List<Column> columns, Row...  rows)` constructor, which you should save for later. Make sure to write tests for the Row class. See lab5 for examples of how to test components of a package. Consider adding a command to your makefile that runs your unit tests, using the optional part of lab5 as a guide.

4. Implement the parts of the Table class needed to create a new Table, add a Row to it, and print an entire Table.

5. Implement the Database class.

6. Implement `insert` and `load`.

7. The right strategy for `select` (not the most efficient, just the easiest) is to implement the kind that starts with '*name* `:`' first, and then implement the other (printing) kind simply by creating an unnamed table and then printing it rather than storing it away.
8. Implement the kind of `select` that takes a single table and has no conditions.
9. Implement the `Row(List<Column> columns, Row... rows)` constructor.
10. Now get single-table `select` with conditions to work.
11. Finally, work on the two-table variety of `select`.
12. In the above steps, as you decide on implementation strategies and the data representations you will use, write them down in your internal documentation. When you introduce new methods or classes, write the comments first.

You *may* throw all our skeleton code away if you prefer. You are not required to implement `Table`, `Row`, etc. However, we strongly recommend that you *don't* do this if your only reason is that you can't figure out how to do it our way. In that case, pester us with questions until you *do* understand.